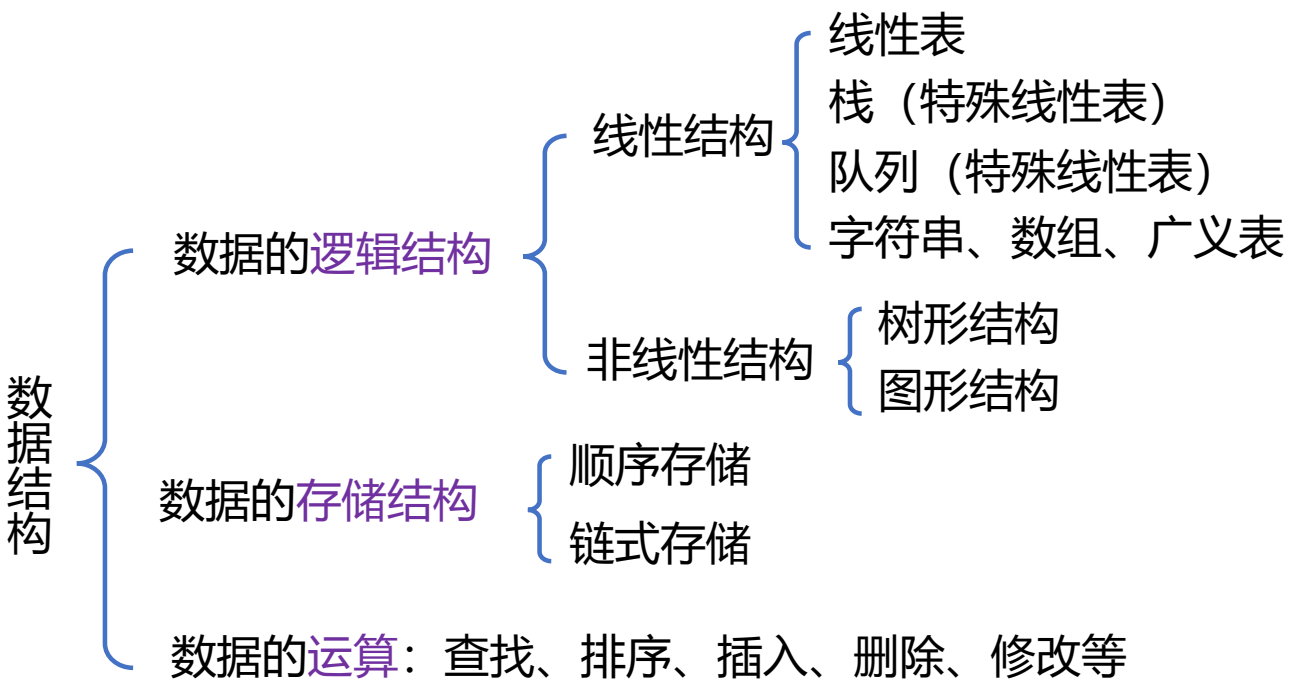
知识回顾


数据结构
数据的逻辑结构
线性结构
线性表
栈（特殊线性表）
队列（特殊线性表）
字符串、数组、广义表
非线性结构
树形结构
图形结构
数据的存储结构
顺序存储
链式存储
数据的运算：查找、排序、插入、删除、修改等

《数据结构》

第9章 查找

杭州电子科技大学
HANGZHOU DIANZI UNIVERSITY

9.1 查找的基本概念

9.2 静态查找表 （线性表的查找）

9.3 动态查找表 （树表的查找）

9.4 哈希表

9.1 查找的基本概念

问题：在哪里找？

————查找表

- 查找表是由同一类型的数据元素（或记录）构成的集合。由于“集合”中的数据元素之间存在着**松散的关系**，因此查找表是一种应用灵便的结构。

准考证号	姓名	各科成绩							总分
		政治	语文	外语	数学	物理	化学	生物	
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
179325	陈红	85	86	88	100	92	90	45	586
179326	陆华	78	75	90	80	95	88	37	543
179327	张平	82	80	78	98	84	96	40	558
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

9.1 查找的基本概念

问题：什么查找？

————根据给定的某个值，在查找表中确定一个其关键字等于给定值的数据元素或（记录）。

- 关键字　　用来标识一个数据元素（或记录）的某个数据项的值
  - 主关键字　　可唯一地标识一个记录的关键字是主关键字
  - 次关键字　　反之，用以标识若干记录的关键字是次关键字

9.1 查找的基本概念

问题：查找成功否

查找————根据给定的某个值，在查找表中确定一个其关键字等于给定值的数据元素或（记录）

- 若查找表中存在这样一个记录，则称“查找成功”。
  - 查找结果给出整个记录的信息，或指示该记录在查找表中的位置；
- 否则称“查找不成功”。
  - 查找结果给出“空记录”或“空指针”。

9.1 查找的基本概念

问题：查找的目的是什么？

对查找表经常进行的操作：

1、查询某个“特定的”数据元素是否在查找表中；

2、检索某个“特定的”数据元素的各种属性；

3、在查找表中插入一个数据元素；

4、删除查找表中的某个数据元素；

9.1 查找的基本概念

问题：查找怎么分类?

查找表可分为两类：

- 静态查找表：

  仅做“查询”（检索）操作的查找表。

- 动态查找表：

  做“插入”和“删除”操作的查找表。

  有时在查询之后，还需要将“查询”结果为“不在查找表中”的数据元素插入到查找表中；或者，从查找表中删除其“查询”结果为“在查找表中”的数据元素，此类表为动态查找表。

9.1 查找的基本概念

杭州电子科技大学
HANGZHOU DIANZI UNIVERSITY

问题：如何评价查找算法?

查找算法的评价指标：

关键字的平均比较次数，也称**平均查找长度**

ASL（Average Search Length）

$ASL = \sum_{i=1}^{n} p_i c_i$ （关键字比较次数的期望）

n：记录的个数

p_i：查找表中第 i 个记录的概率（通常认为$p_i = 1/n$）

c_i：找到第i个记录所需的比较次数

9.1 查找的基本概念

杭州电子科技大学
HANGZHOU DIANZI UNIVERSITY

问题：查找过程中我们要研究什么？

查找的方法取决于查找表的结构，即表中数据元素是依何种关系组织在一起的。

由于对查找表来说，在集合中查询或检索一个“特定的”数据元素时，若无规律可循，只能对集合中的元素一一加以辨认直至找到为止。

而这样的“查询”或“检索”是任何计算机应用系统中使用频度都很高的操作，因此设法提高查找表的查找效率，是本章讨论问题的出发点。

为提高查找效率，一个办法是在构造查找表时，在集合中的数据元素之间人为地加上某种确定的约束关系。

研究查找表的各种组织方法及其查找过程的实施。

第9章 查找

9.2 线性表的查找

杭州电子科技大学
HANGZHOU DIANZI UNIVERSITY

一、顺序查找（线性查找）

二、折半查找（二分或对分查找）

三、分块查找（索引查找）

9.2.1 顺序查找

应用范围：

- 顺序表或线性链表表示的静态查找表
- 表内元素之间无序

顺序表的表示：

数据元素类型定义：

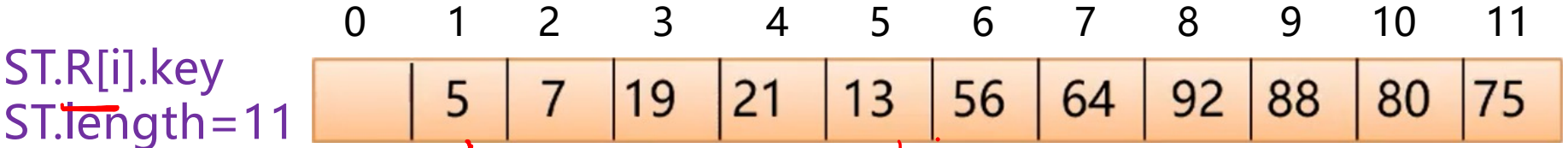
```
typedef struct {
    KeyType key;    //关键字域
    ......          //其他域
} ElemType;
```

```
typedef struct {//顺序表结构类型定义
    ElemType  *R;  //表基址
    int  length;     //表长
}SSTable;  //Sequential Search Table
SSTable  ST;   //定义顺序表ST
```

9.2.1 顺序查找

在顺序表ST中查找值为key的数据元素　　从最后一个元素开始比较

例如：查找13： 找到，返回5　　查找60：未找到，返回0




```
int Search_Seq( SSTable ST , KeyType key ){
        //若成功返回其位置信息，否则返回0
    for( i=ST.length; i>=1; - - i )
        if ( ST.R[ i ].key==key )  return i;
    return 0;
}
```

9.2.1 顺序查找

其他形式：

```
int Search_Seq(SSTable ST,  KeyType key) {
    for (i = ST.length ;  ST.R[i].key != key ;  - - i )
        if ( i <= 0) break ;
    if (i > 0) return i ;
    else return 0 ;
}
```

9.2.1 顺序查找

其他形式：

```
int Search_Seq(SSTable ST,  KeyType key) {
    for (i = ST.length ;  ST.R[i].key != key ;  - - i )
        if ( i <= (
    if (i > 0) retu
    else return (
}
```

每执行一次循环都要进行两次比较，是否能改进?

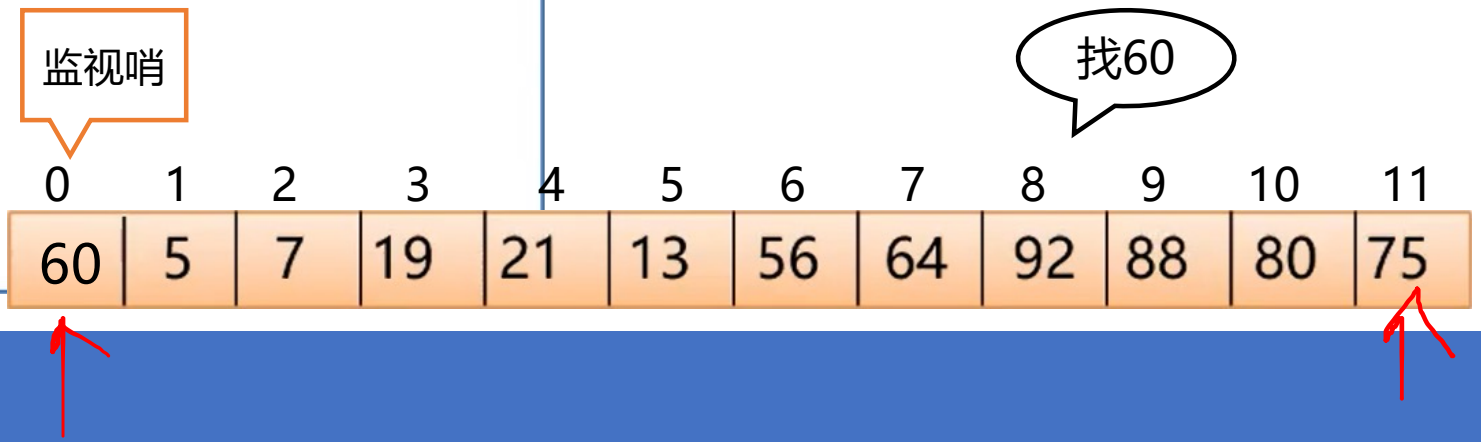
```
int Search_Seq(SSTable ST,  KeyType key) {
    for (i = ST.length ;  ST.R[i].key != key && i>0 ;  - - i );
    if (i > 0) return i ;
    else return 0 ;
}
```

9.2.1 顺序查找

改进：把待查关键字key存入表头（“哨兵”、“监视哨”），从而避免逐个比较，可免去查找过程中每一步都要检测是否查找完毕，加快速度

```
int Search_Seq(SSTable ST,  KeyType key) {
      ST.R[0].key=key;
  for (i = ST.length ;  ST.R[i].key != key ;  - - i ) ;

          return i ;
}
```



9.2.1 顺序查找

【算法9.1】设置监视哨的顺序查找

```
int Search_Seq( SSTable  ST , KeyType  key ){
      ST.R[0].key =key;
      for( i=ST.length; ST.R[ i ].key!=key;  - - i  );
      return i;
}
```

当ST.length较大时，此改进能使进行一次查找所需的平均时间几乎减少一半。

9.2.1 顺序查找

【算法9.1】时间效率分析

```
int Search_Seq( SSTable  ST , KeyType  key ){
      ST.R[0].key =key;
      for( i=ST.length; ST.R[ i ].key!=key;  - - i  );
      return i;
}
```

比较次数与key位置有关：

- 查找第i个元素，需比较n-i+1次
- 查找失败，需比较n+1次

	0	1	2	3	4	5	6	7	8	9	10	11
	哨兵	5	7	19	21	13	56	64	92	88	80	75
比较次数	12	11	10	9	8	7	6	5	4	3	2	1

顺序查找的性能分析

- 时间复杂度：O(n)

  查找成功时的平均查找长度，设表中各记录查找概率相等

  $$ASL(n) = \frac{1}{n}\sum_{i=1}^{n}(n-i+1) = (1+2+\ldots+n)/n = (n+1)/2$$

- 空间复杂度：一个辅助空间——O(1)

顺序查找的性能分析

- 查找不成功情况不能忽视时的平均查找长度

  假设查找成功和不成功的可能性相同，对每个记录的查找概率也相等，则概率为1/(2n)

$$ASL^{*}(n) = \frac{1}{2n}\sum_{i=1}^{n}(n-i+1) + \frac{1}{2}(n+1) = 3(n+1)/4$$

讨论

1.记录的查找概率不相等如何提高查找效率?

查找表存储记录原则——按查找次数高低存储

1）查找概率越高，存放位置越靠后，比较次数越少；

2）查找概率越低，存放位置越靠前，比较次数越多。

2.记录的查找概率无法测定时如何提高查找效率?

方法——按查找概率动态调整记录顺序

1）在每个记录中设一个访问频度域；

2）始终保持记录按非递减有序的次序排列；

3）每次查找后均将刚查到的记录直接移到表尾。

顺序查找的特点

杭州电子科技大学
HANGZHOU DIANZI UNIVERSITY

优点：算法简单，逻辑次序无要求，且不同存储结构均适用。

缺点：ASL太长，时间效率太低。

9.2 线性表的查找

杭州电子科技大学
HANGZHOU DIANZI UNIVERSITY

一、顺序查找（线性查找）

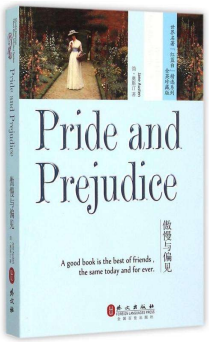
二、折半查找（二分或对分查找）

三、分块查找（索引查找）

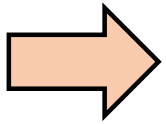
顺序查找的特点

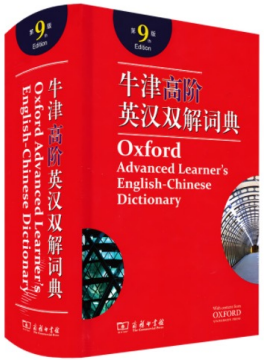
优点：算法简单，逻辑次序无要求，且不同存储结构均适用。

缺点：ASL太长，时间效率太低。



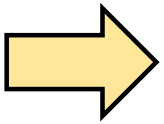
单词无序





单词按字母顺序排序

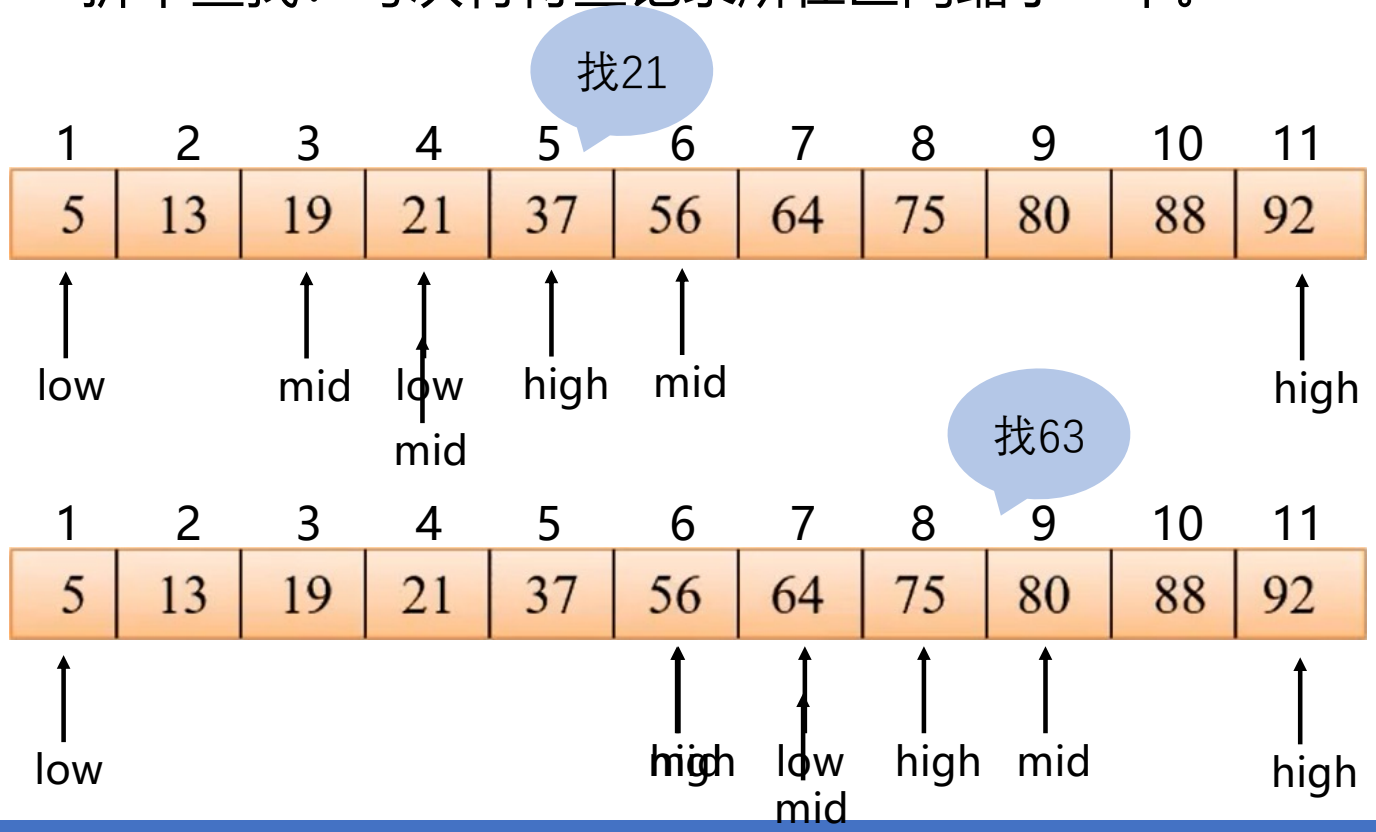
有序表表示静态查找表



折半查找

9.2.2 折半查找

折半查找：每次将待查记录所在区间缩小一半。



mid = (low+high)/2

key<mid 则：high=mid-1

key>mid 则：low=mid+1

key==mid，找到

high<low，结束

9.2.2 折半查找

折半查找算法：（非递归算法）

- 设表长为n，low、high和mid分别指向待查元素所在区间的上界、下界和中点，key为给定的要查找的值：
- 初始时，令low=1，high=n，mid=$\lfloor(low + high)/2\rfloor$
- 让key与mid指向的记录比较
  - ✓ 若key==R[mid].key，查找成功
  - ✓ 若key<R[mid].key，则high=mid-1
  - ✓ 若key>R[mid].key，则low=mid+1
- 重复上述操作，直至low>high时，查找失败

9.2.2 折半查找

杭州电子科技大学
HANGZHOU DIANZI UNIVERSITY

【算法9.2】折半查找

```
int Search_Bin ( SSTable ST,  KeyType key ) {
   low = 1 ; high = ST.length ;        // 置区间初值
   while (low <= high) {
        mid = (low + high) / 2 ;
        if (ST.R[mid].key == key)   return mid ;    // 找到待查元素
        else  if (key < ST.R[mid].key) //缩小查找区间
                    high=mid-1;          //继续在前半区间进行查找
              else low = mid + 1;      // 继续在后半区间进行查找
    }
   return 0 ;          // 顺序表中不存在待查元素
} // Search_Bin
```

9.2.2 折半查找

杭州电子科技大学
HANGZHOU DIANZI UNIVERSITY

补充【算法9.2】折半查找——递归算法

```
int Search Bin (SSTable ST, keyType key, int low, int high){
    if(low>high) return 0;    //查找不到时返回0
    mid = (low+high)/2;
    if(key == ST.elem[mid].key) return mid;
    else if(key<Stelem[mid].key)
        return Search Bin(ST, key, low, mid-1); // 递归，在前半区间进行查找
    else
        return Search Bin(ST, key, mid+1, high); // 递归，在后半区间进行查找
}
```

折半查找的性能分析——判定树

杭州电子科技大学
HANGZHOU DIANZI UNIVERSITY

查找成功：

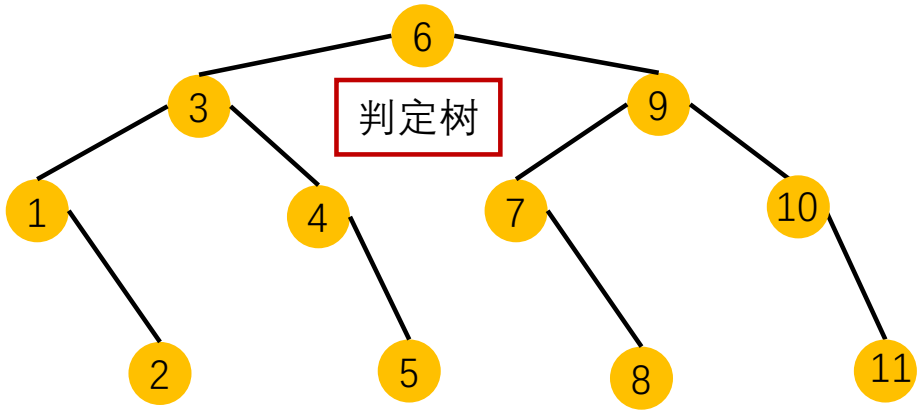
比较次数 = 路径上的节点数

比较次数 = 节点的层数

比较次数<=树的深度

$= \lfloor \log_2 n \rfloor + 1$

i	1	2	3	4	5	6	7	8	9	10	11
	5	13	19	21	37	56	64	75	80	88	92
C_i	3	4	2	3	4	1	3	4	2	3	4



圆形——内结点
代表查找成功的情况

折半查找的性能分析——判定树

i	1	2	3	4	5	6	7	8	9	10	11
	5	13	19	21	37	56	64	75	80	88	92
C_i	3	4	2	3	4	1	3	4	2	3	4

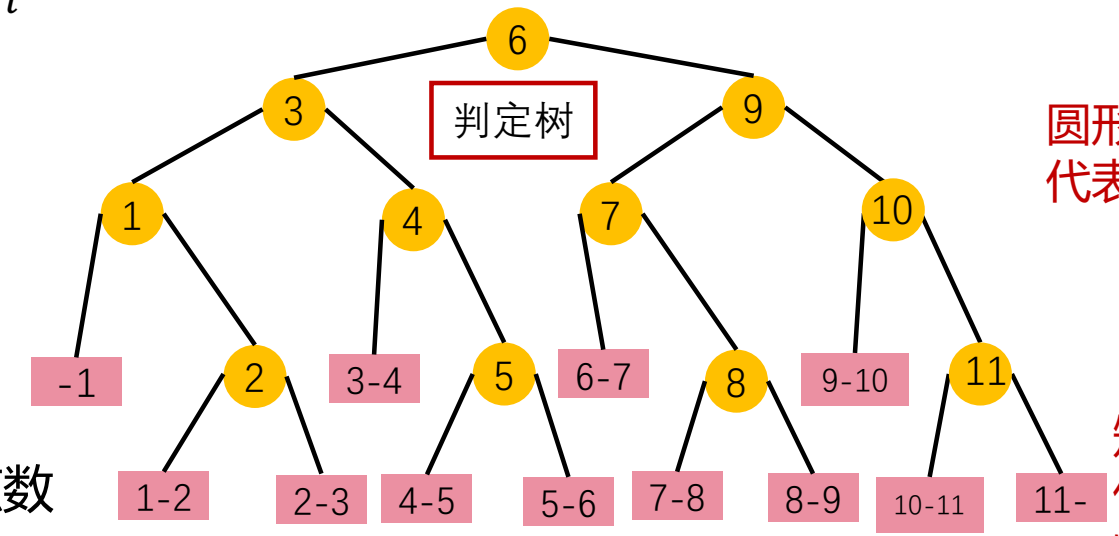
查找成功：

比较次数 = 路径上的节点数

比较次数 = 节点的层数

比较次数<=树的深度

$= \lfloor \log_2 n \rfloor + 1$



判定树

圆形——内结点
代表查找成功的情况

查找不成功：

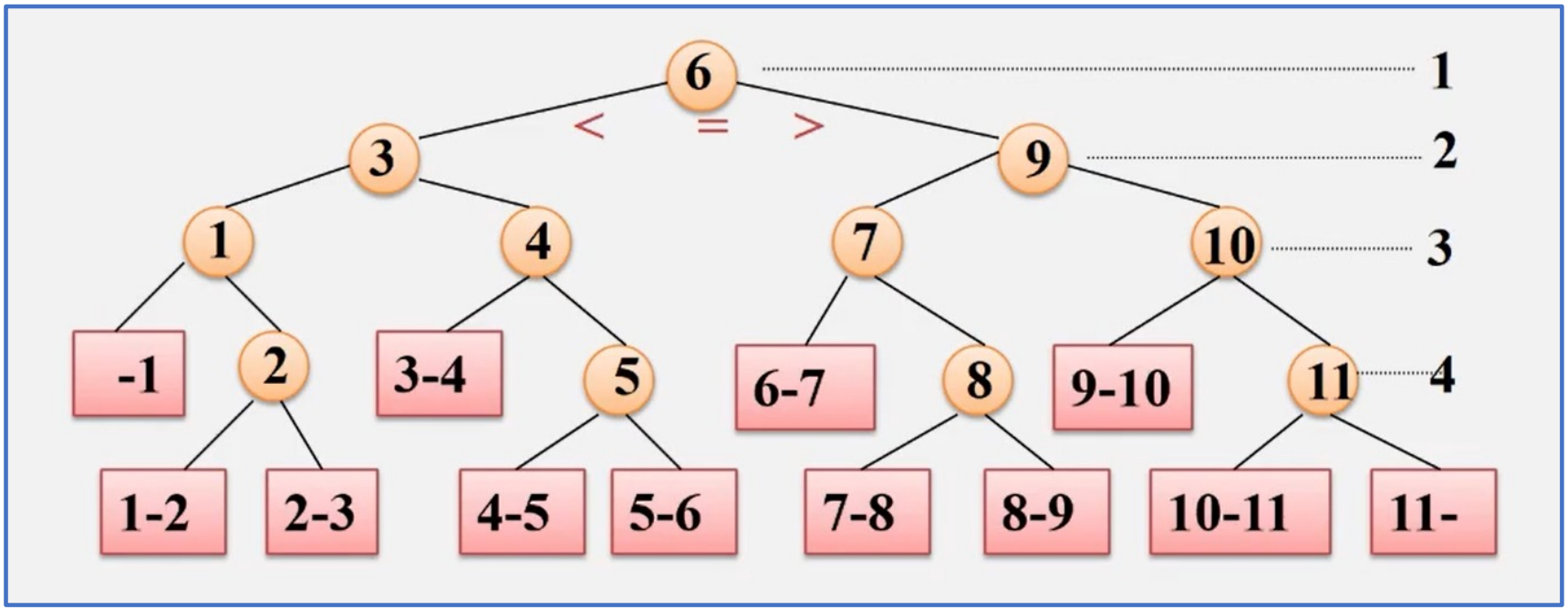
比较次数 = 路径上的内部节点数

比较次数<=$\lfloor \log_2 n \rfloor + 1$

矩形——外结点
代表查找不成功的情况

练习

- 假定每个元素的查找概率相等，求查找成功时的平均查找长度



ASL = (1*1+2*2+4*3+4*4) /11 = 33/11 = 3

折半查找的性能分析

平均查找长度ASL（成功时）：

设表长 n = $2^h - 1$, 则h = $log_2(n+1)$,此时，判定树为深度 = h的满二叉树，且表中每个记录的查找概率相等：$P_i = 1/n$

第 j 层的每个结点要比较的次数

则：$ASL_{bs} = \sum_{i=1}^{n} P_i C_i = (1/n)\sum_{i=1}^{n} C_i = (1/n)\sum_{j=1}^{h} j * 2^{j-1}$

第 j 层结点数

第 j 层要比较的总次数

$= ((n+1)/n)\ log_2(n+1) - 1$

$\approx log_2(n+1) - 1\ \ (n > 50)$

折半查找的性能分析

折半查找优点：效率比顺序查找高

折半查找缺点：只适用于有序表，且限于顺序存储结构（对线性链表无效）。

9.2 线性表的查找

杭州电子科技大学
HANGZHOU DIANZI UNIVERSITY

一、顺序查找（线性查找）

二、折半查找（二分或对分查找）

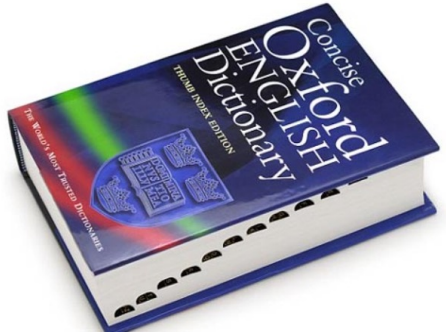
三、分块查找（索引查找）

折半查找的性能分析

折半查找优点：效率比顺序查找高

折半查找缺点：只适用于有序表，且限于顺序存储结构（对线性链表无效）。

单词按字母顺序排序



——分块查找
（索引顺序表的查找）

9.2.3 分块查找（索引顺序查找）

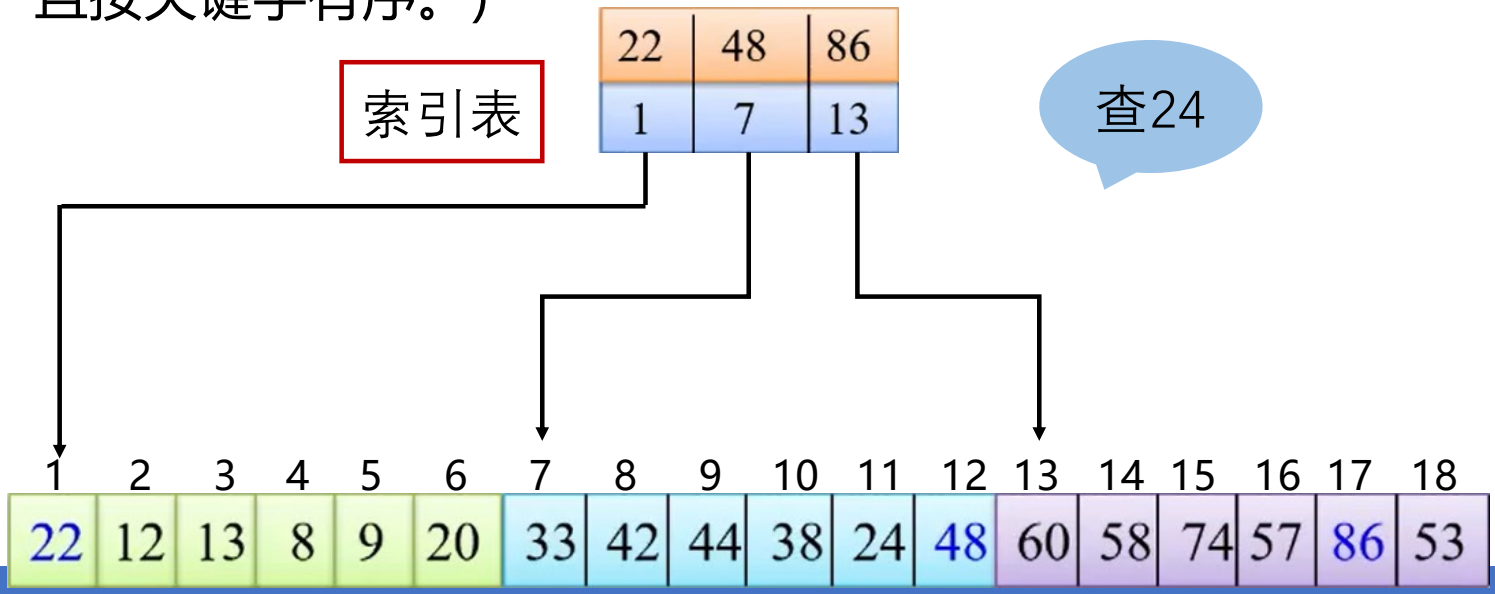
条件：1、将表分成几块，且表或者有序，或者分块有序；

若i<j，则第j块中所有记录的关键字均大于第i块中的最大关键字

2、建立“索引表”（每个节点含有最大关键字域和指向本块第一个节点的指针，且按关键字有序。）

查找过程：先确定待查记录所在块（顺序或折半查找）

再在块内查找（顺序查找）



分块查找性能分析

查找效率：ASL = $L_b + L_w$

对索引表查找的ASL

对块内查找的ASL

$$ASL_{bs} \approx \log_2(\frac{n}{s}+1)+\frac{s}{2} \qquad (\log_2 n \le ASL_{bs} \le \frac{n+1}{2})$$

S为每块内部的记录个数，n/s即块的数目

例如：当n=9,s=3时，$ASL_{bs} = 3.5$ 而折半法为3.1，顺序法为5

分块查找优缺点

- 优点：插入和删除比较容易，无需进行大量移动
- 缺点：要增加一个索引表的存储空间并对初始索引表进行排序运算
- 适用情况：如果线性表既要快速查找又经常动态变化，则可采用分块查找

查找方法比较

	顺序查找	折半查找	分块查找
ASL	最大	最小	中间
表结构	有序表、无序表	有序表	分块有序
存储结构	顺序表、线性链表	顺序表	顺序表、线性链表

第9章 查找

杭州电子科技大学
HANGZHOU DIANZI UNIVERSITY

9.1 查找的基本概念

9.2 静态查找表 （线性表的查找）

9.3 动态查找表 （树表的查找）

9.4 哈希表

9.3 树表的查找

杭州电子科技大学
HANGZHOU DIANZI UNIVERSITY

当表插入、删除操作频繁时，为维护表的有序性，需要移动表中很多记录。

改用动态查找表——几种特殊的树

表结构在查找过程中动态生成

对于给定值key

若表中存在，则成功返回；

否则，插入关键字等于Key的记录

二叉排序树

平衡二叉树

B树（B-tree）

B^+树

键树

红黑树

9.3 树表的查找

二叉排序树（Binary Sort Tree）又称二叉搜索树、二叉查找树

定义：

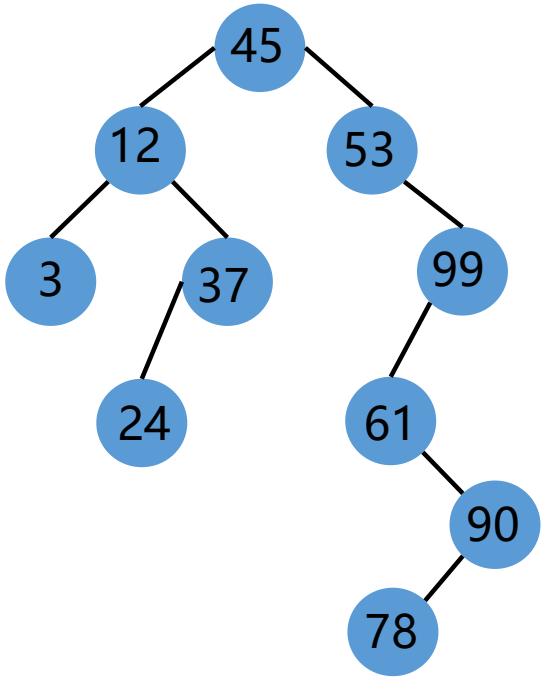
二叉排序树或是空树，或是满足如下性质的二叉树：

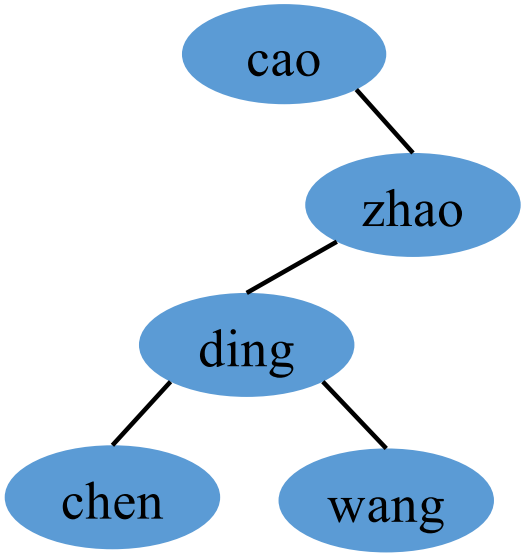
（1）若其左子树非空，则左子树上所有结点的值均小于根结点的值；

（2）若其右子树非空，则右子树上所有结点的值均大于等于根结点的值；

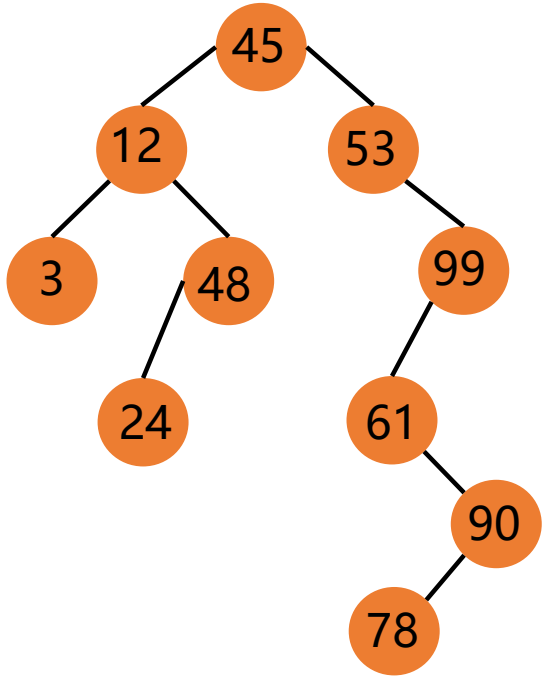
（3）其左右子树本身又各是一棵二叉排序树

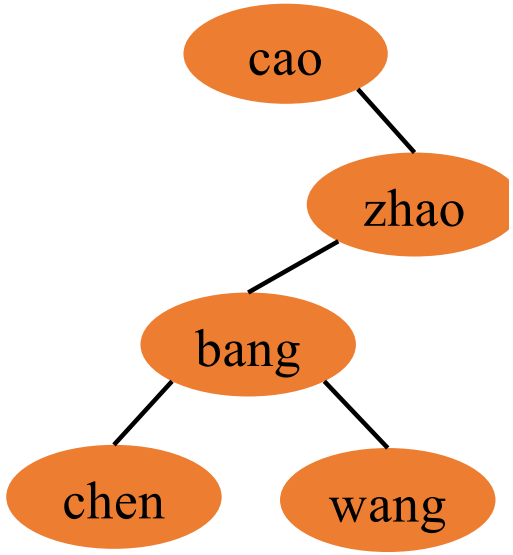
二叉排序树





二叉排序树



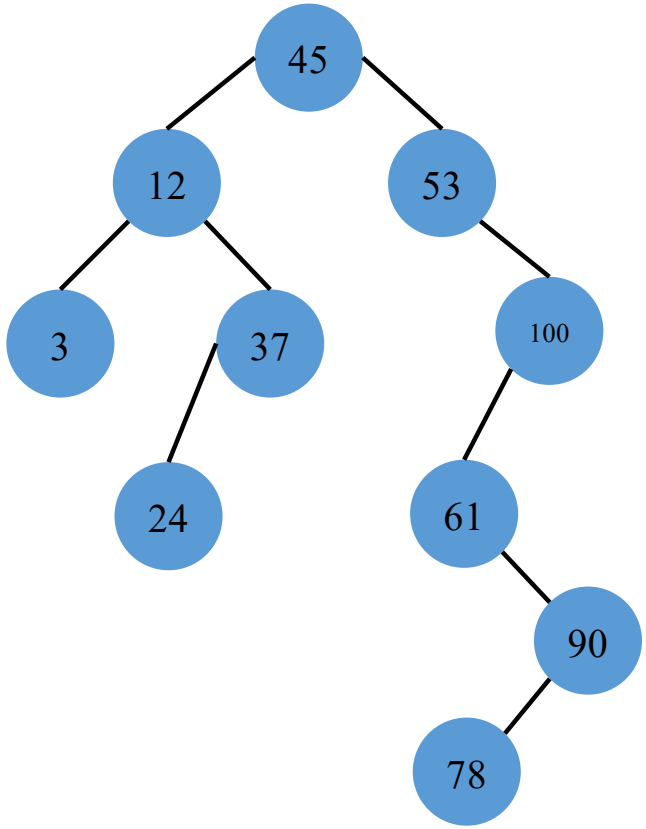


非二叉排序树

思考

- 中序遍历二叉树
- 结果有什么规律?

3 12 24 37 53 61 78 90 100 → 递增



二叉排序树的性质:

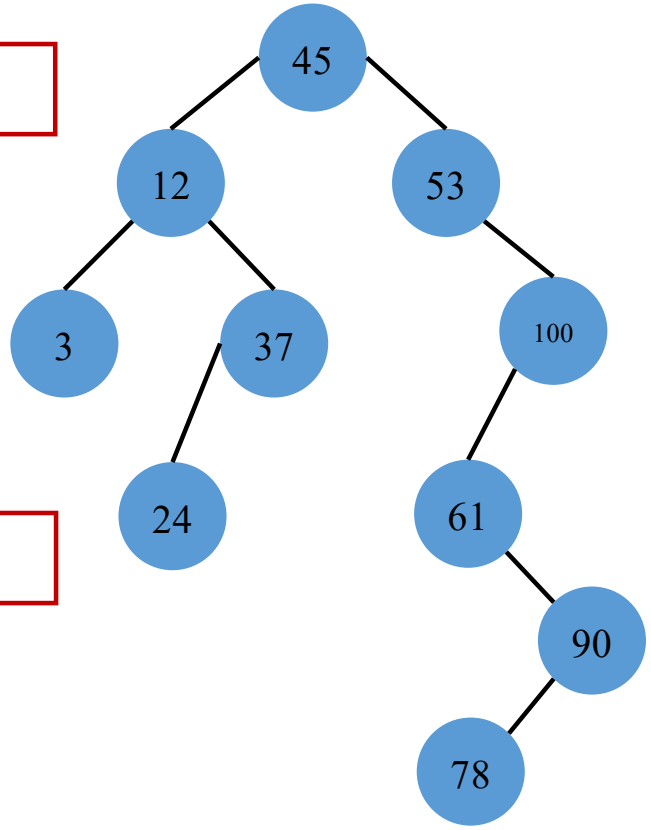
- 中序遍历非空的二叉排序树所得到的数据元素序列是一个按关键字排列的递增有序序列。

二叉排序树的操作——查找

- 若查找的关键字等于根节点，成功
- 否则
  - 若小于根节点，查其左子树
  - 若大于根节点，查其右子树
- 在左右子树上的操作类似

查找24

查找60



二叉排序树的存储结构

杭州电子科技大学
HANGZHOU DIANZI UNIVERSITY

```
typedef struct {
      KeyType key;                  //关键字项
      InfoType otherinfo;           //其他数据域
} ElemType;
```

```
typedef struct BSTNode {
    ElemType  data;                          //数据域
    struct BSTNode  *lchild, *rchild;        //左右孩子指针
}  BSTNode, *BSTree;

BSTree T;  //定义二叉排序树T
```

算法9.5 二叉排序树的递归查找

【算法思想】

（1）若二叉排序树为空，则查找失败，返回空指针。

（2）若二叉排序树非空，将给定值key与根节点的关键字T->data.key进行比较

①若key等于T->data.key,则查找成功，返回根节点地址

②若key小于T->data.key,则进一步查找左子树

③若key大于T->data.key,则进一步查找右子树

算法9.5 二叉排序树的递归查找

【算法描述】

```
BSTree SearchBST(BSTree T,KeyType key) {
  if((!T) || key==T->data.key) return T;
  else if (key<T->data.key)
          return SearchBST(T->lchild,key);//在左子树中继续查找
       else return SearchBST(T->rchild,key); //在右子树中继续查找
} // SearchBST
```

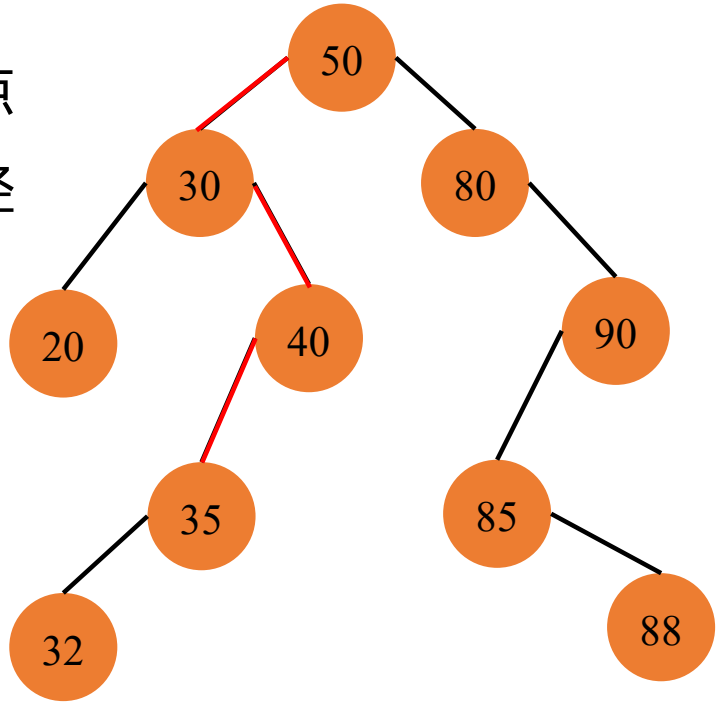
二叉排序树的查找分析

杭州电子科技大学
HANGZHOU DIANZI UNIVERSITY

二叉排序树上查找某关键字等于给定值的结点过程，其实就是走了一条从根到该结点的路径

比较的关键字次数 = 此节点所在的层次数

最多的比较次数 = 树的深度

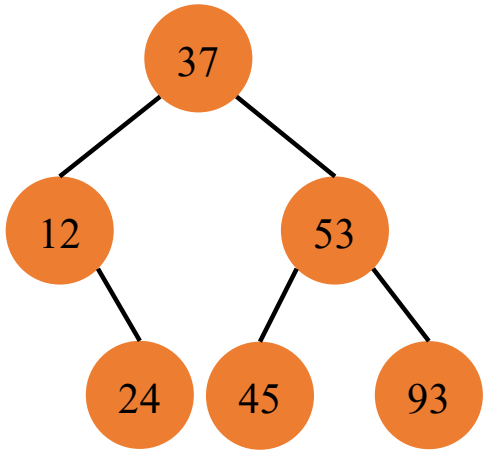


查找关键字：35

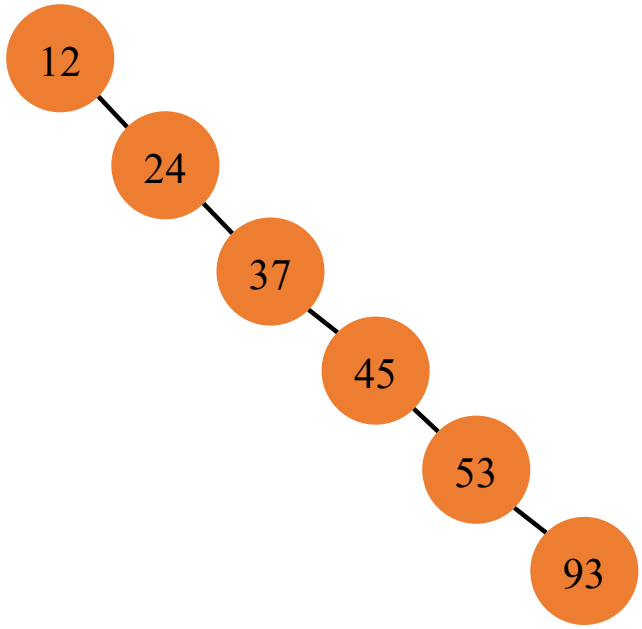
二叉排序树的查找分析

二叉排序树的平均查找长度：

例（12，24，37，45，53，93）



ASL = （1+2+2+3+3+3）/6 = 14/6



ASL = （1+2+3+4+5+6）/6 = 21/6

二叉排序树的查找分析

含有 n 个结点的二叉排序树的平均查找长度和树的形态有关

最好情况：初始序列{45，24，53，12，37，93}

ASL = $log_2(n+1)-1$;

树的深度为：$\lfloor log_2 n \rfloor$+1（向下取整）

与折半查找中的判定树相同

（形态比较均衡）：O($log_2 n$)

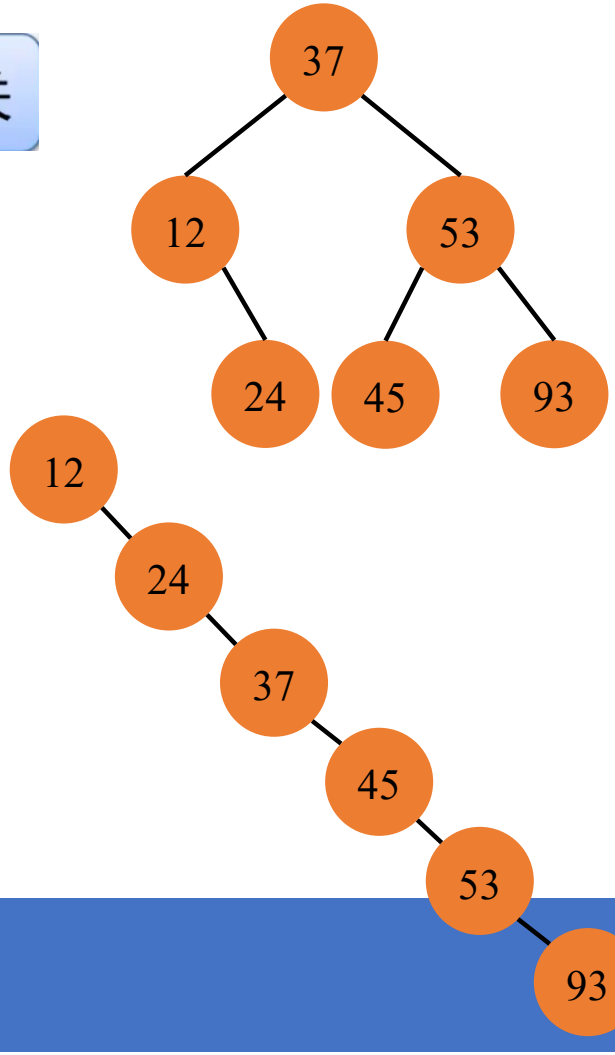
最坏情况：初始形态{ 12，24，37，45，53，93}

插入的n个元素从一开始就有序

——变成单只树的形态

此时树的深度为n，ASL = (n+1) /2

查找效率与顺序查找情况相同：O(n)



二叉排序树

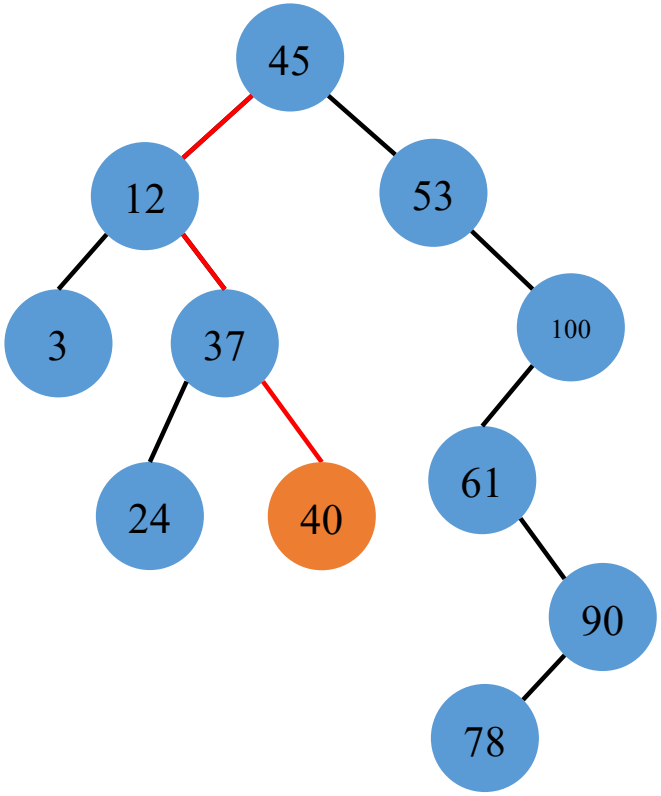
杭州电子科技大学
HANGZHOU DIANZI UNIVERSITY

问题：如何提高形态不均衡的二叉排序树的查找效率？

解决办法：做“平衡化”处理，即尽量让二叉树的形状均衡！

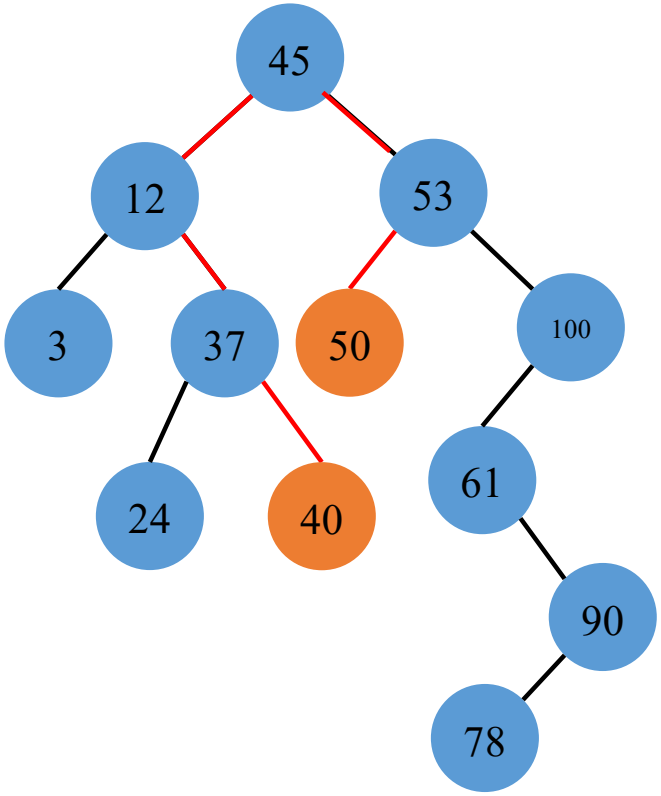
平衡二叉树

二叉排序树的操作——插入



插入40， 是37的右孩子

二叉排序树的操作——插入



插入40， 是37的右孩子

插入50， 是53的左孩子

二叉排序树的操作——插入

- 若二叉排序树为空，则插入节点作为根结点插入到空树中
- 否则，继续在其左、右子树上查找
  - 树中已有，不再插入
  - 树中没有
    - 查找直到某个结点的左子树或右子树为空为止，则插入结点应为该结点的左孩子或右孩子

插入的结点一定叶子结点

算法9.6 二叉排序树的插入算法

杭州电子科技大学
HANGZHOU DIANZI UNIVERSITY

【算法描述】

```
Status SearchBST(BiTree T, KeyType key, BiTree f, BiTree &p){
    if (!T) {p=f; return FALSE;}                          //查找不成功
    else  if (key==T->data.key) {p=T; return TRUE;}   //查找成功
    else  if (key<T->data.key)
          return SearchBST (T->lchild, key, T, p);   //在左子树中继续查找
    else  return SearchBST (T->rchild, key, T, p);  //在右子树中继续查找
}//SearchBST
```

算法9.6 二叉排序树的插入算法

杭州电子科技大学
HANGZHOU DIANZI UNIVERSITY

【算法描述】

```
Status InsertBST(BiTree &T, ElemType e){
    if (! SearchBST (T, e.key, NULL, p)) {     //查找不成功
        s=(BiTree) malloc (sizeof (BiTNode));
        s->data=e;    s->lchild=s->rchild=NULL;
        if (!p) T=s;                                  //被插入的结点为新的根结点
        else  if (e.key<p->data.key)  p->lchild=s;  //被插入的结点为左孩子
        else  p->rchild=s;          //被插入的结点为右孩子
        return TRUE; }
     else return FALSE; //树里有与关键字相同的结点，不再插入
}// InsertBST
```

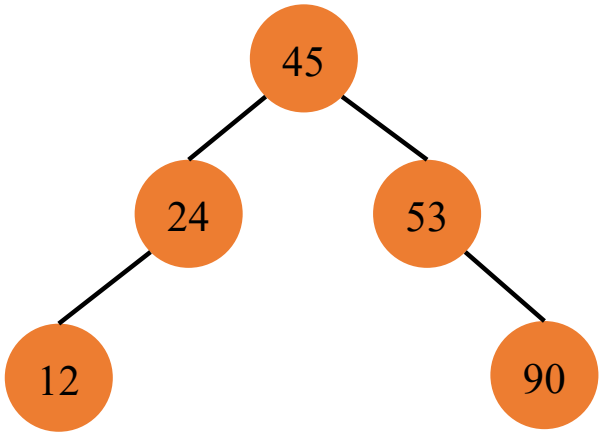
二叉排序树的操作——生成

从空树出发，经过一系列的查找、插入操作之后，可生成一颗二叉排序树。、

例如：设查找的关键字序列为

{ 45， 24， 53， 45， 12， 24， 90 }

可生成二叉排序树如下：



二叉排序树的操作——生成

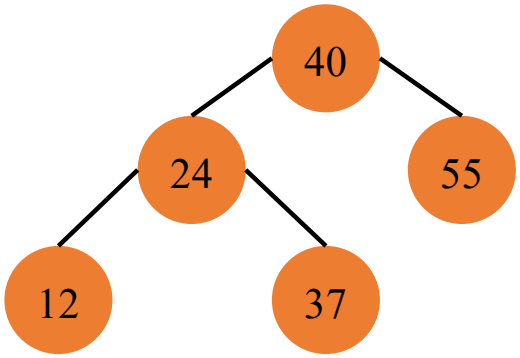
- 一个无序序列可通过构造二叉排序树而变成一个有序序列。构造树的过程就是对无序序列进行排序的过程
- 插入的节点均为叶子节点，故无需移动其他节点。相当于在有序序列上插入记录而无需移动其他记录。
- 但是：

  关键字的输入顺序不同，建立的不同二叉排序树。

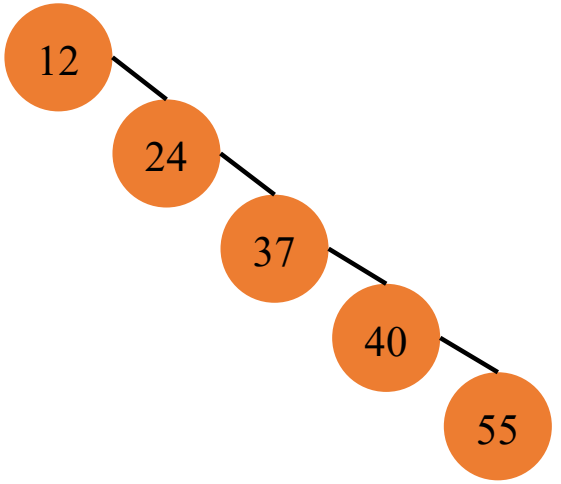
二叉排序树的操作——生成

不同插入次序的序列生成不同形态的二叉排序树

{ 40， 24， 12， 37， 55 }



{ 12， 24， 37， 40， 55 }



二叉排序树的操作——删除

从二叉排序树中删除一个结点，不能把以该结点为根的子树都删去，只能删除该结点，并且还应保证删除后所得的二叉树仍然满足二叉排序树的性质不变。

由于中序遍历二叉排序树可以得到一个递增有序的序列。那么，在二叉排序树中删去一个结点相当于删去有序序列中的一个结点。

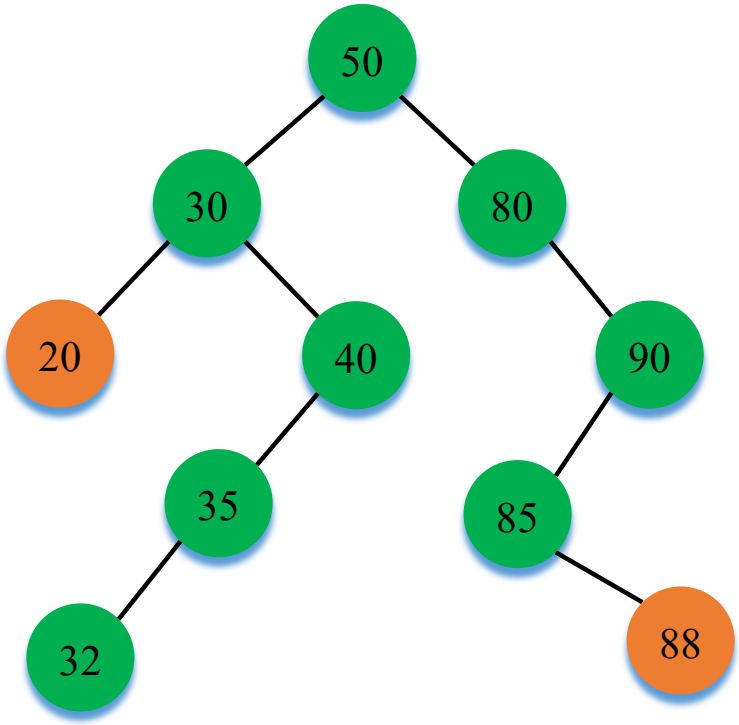
（1）将因删除结点而断开的二叉链表重新链接起来

（2）防止重新链接后的树的高度增加

二叉排序树的操作——删除

（1）被删除的结点是叶子结点：直接删去该结点

例如：



被删结点：20

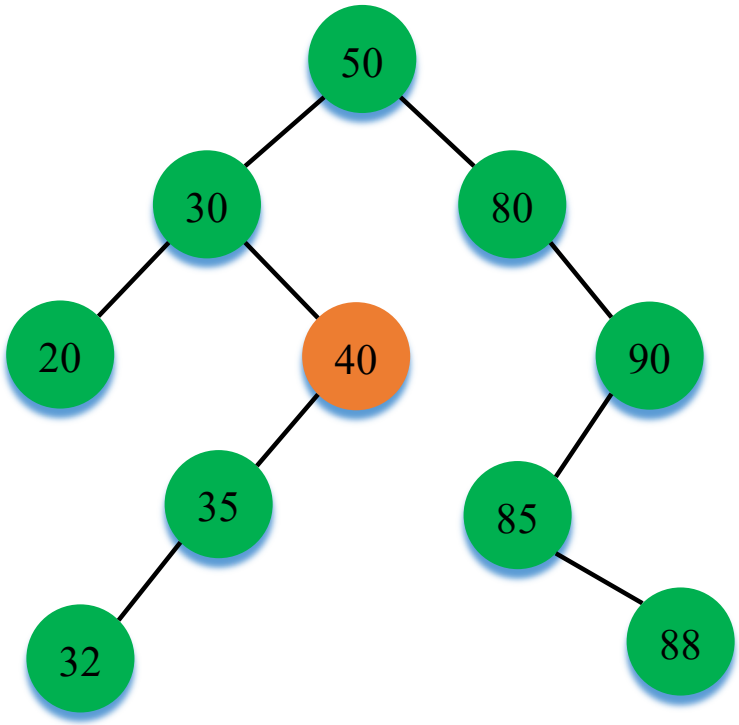
被删结点：88

其双亲结点中相应指针域的值改为空

二叉排序树的操作——删除

（2）被删除的结点是只有左子树或者右子树，用其左子树或者右子树替换它（结点替换）

例如：

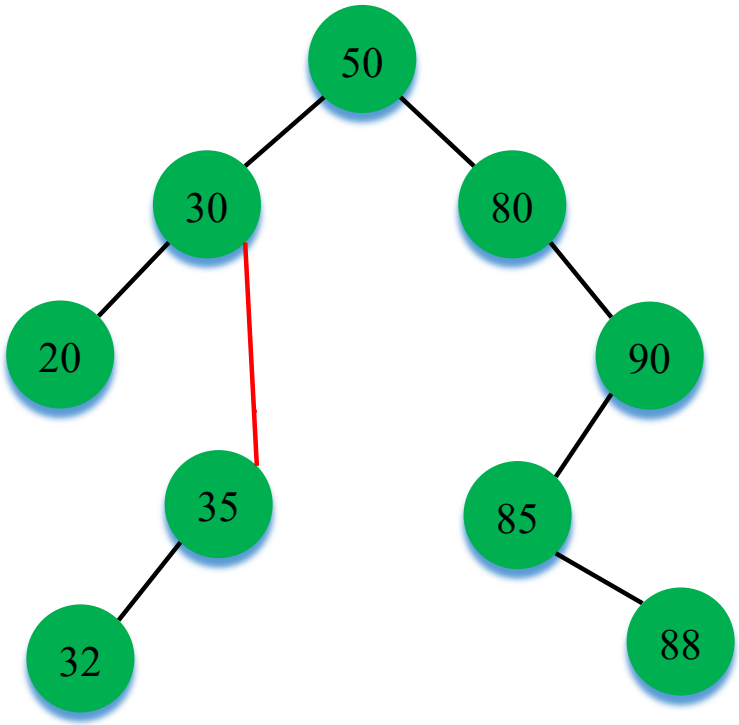


被删结点：40

二叉排序树的操作——删除

（2）被删除的结点是只有左子树或者右子树，用其左子树或者右子树替换它（结点替换）

例如：

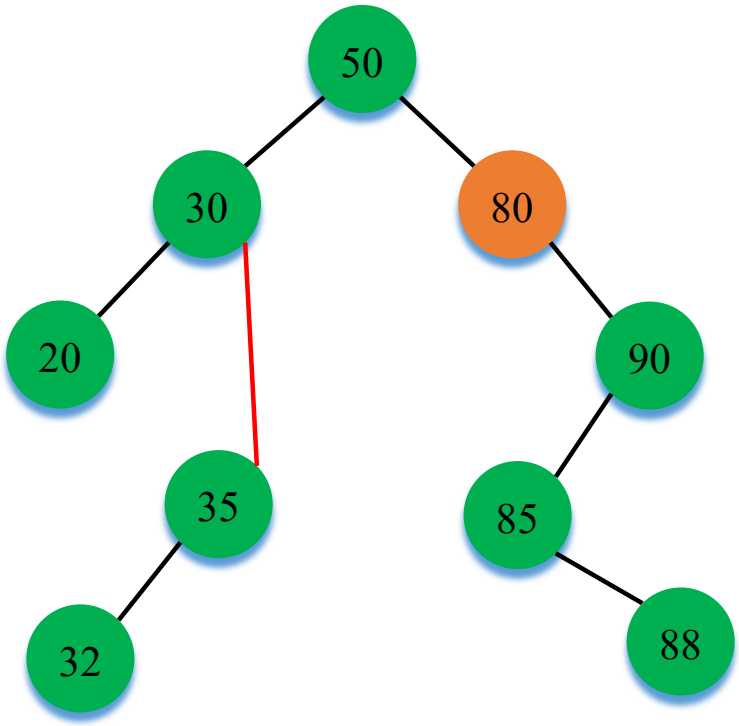


被删结点：40

二叉排序树的操作——删除

（2）被删除的结点是只有左子树或者右子树，用其左子树或者右子树替换它（结点替换）

例如：



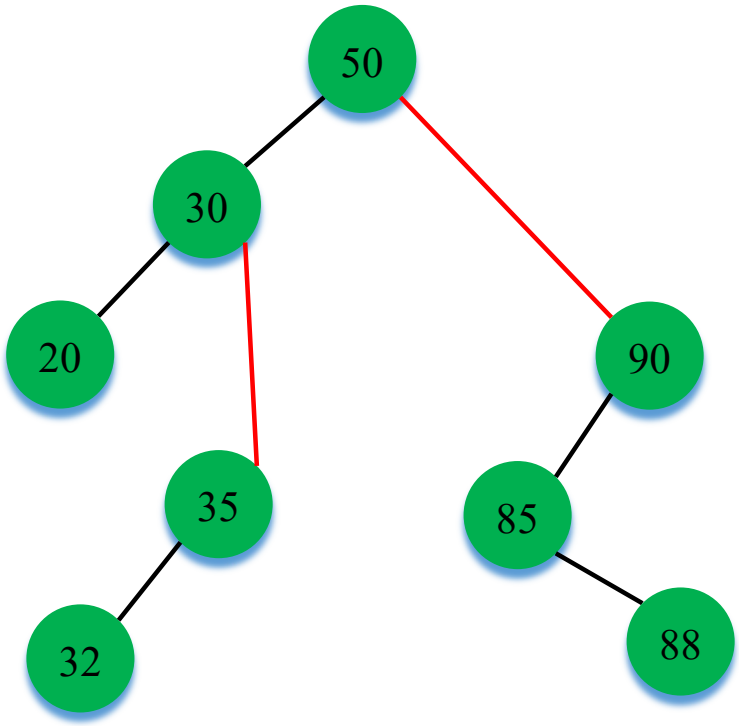
被删结点：40

被删结点：80

二叉排序树的操作——删除

（2）被删除的结点是只有左子树或者右子树，用其左子树或者右子树替换它（结点替换）

例如：



被删结点：40

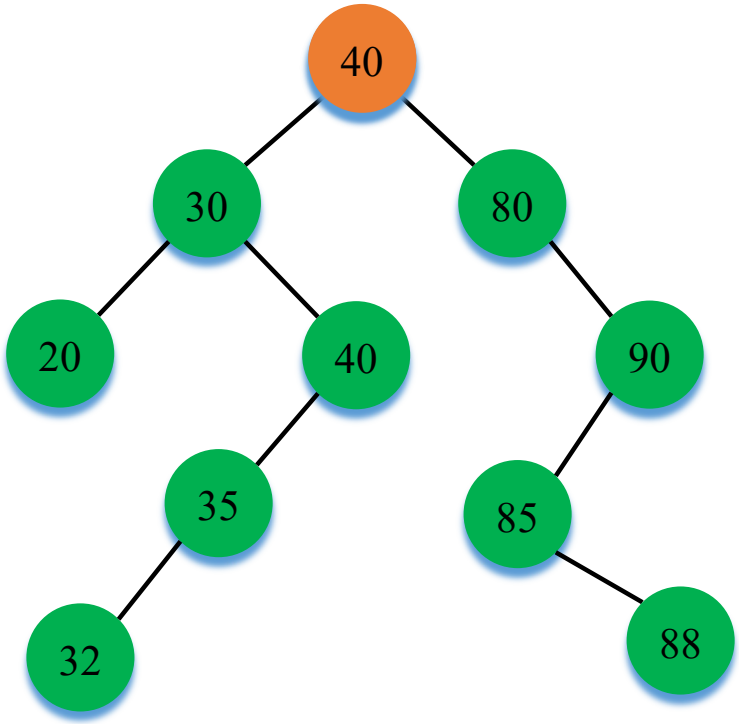
被删结点：80

其双亲结点中相应指针域的值改为“指向被删除结点的左子树或右子树”

二叉排序树的操作——删除

（3）被删除的节点既有左子树，也有右子树

例如：

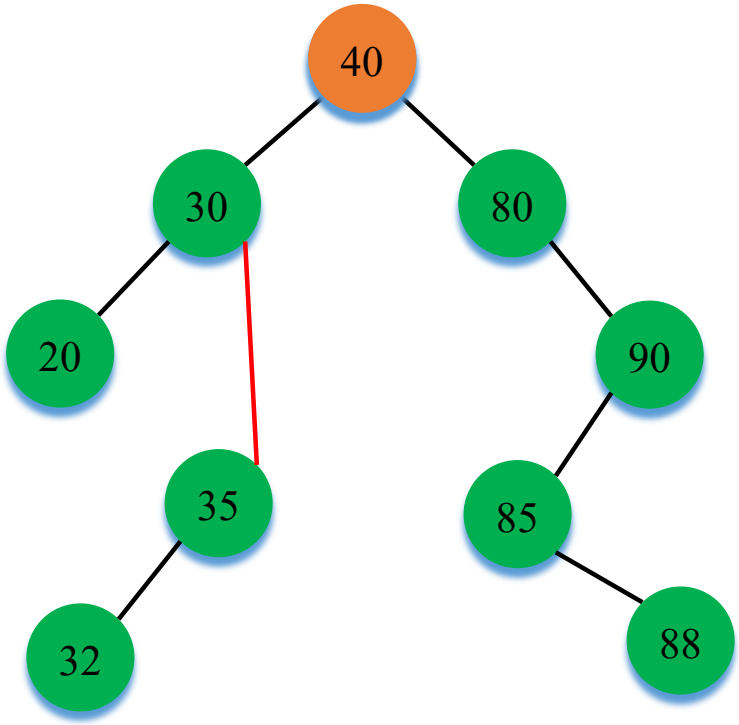


被删结点：50

二叉排序树的操作——删除

（3）被删除的节点既有左子树，也有右子树

例如：



被删结点：50

- 以其中序前驱值替换之（值替换），然后再删除该前驱节点。

  前驱是左子树中最大的节点。

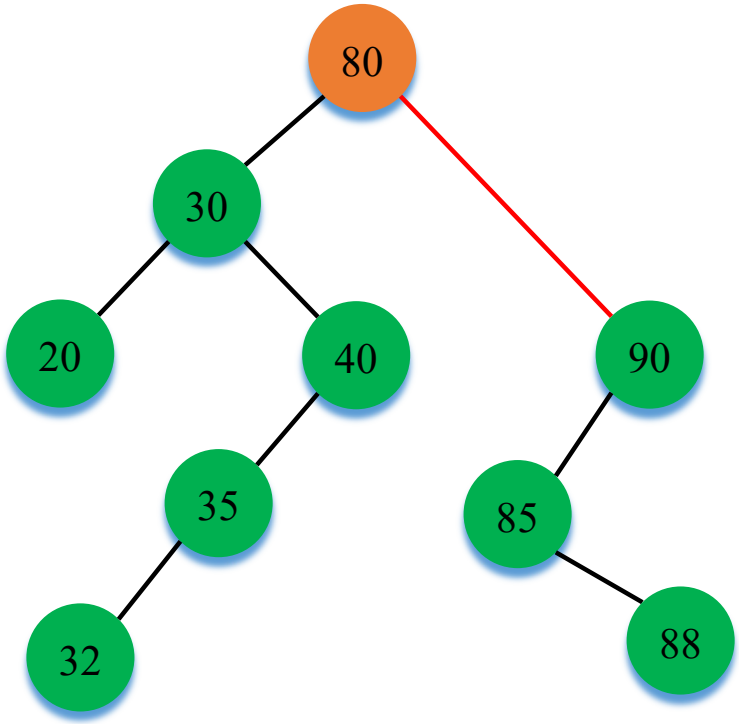
- 也可以用其后继替换之（值替换），然后再删除该后继节点。

  后继是右子树中最小的节点。

二叉排序树的操作——删除

（3）被删除的节点既有左子树，也有右子树

例如：



被删结点：50

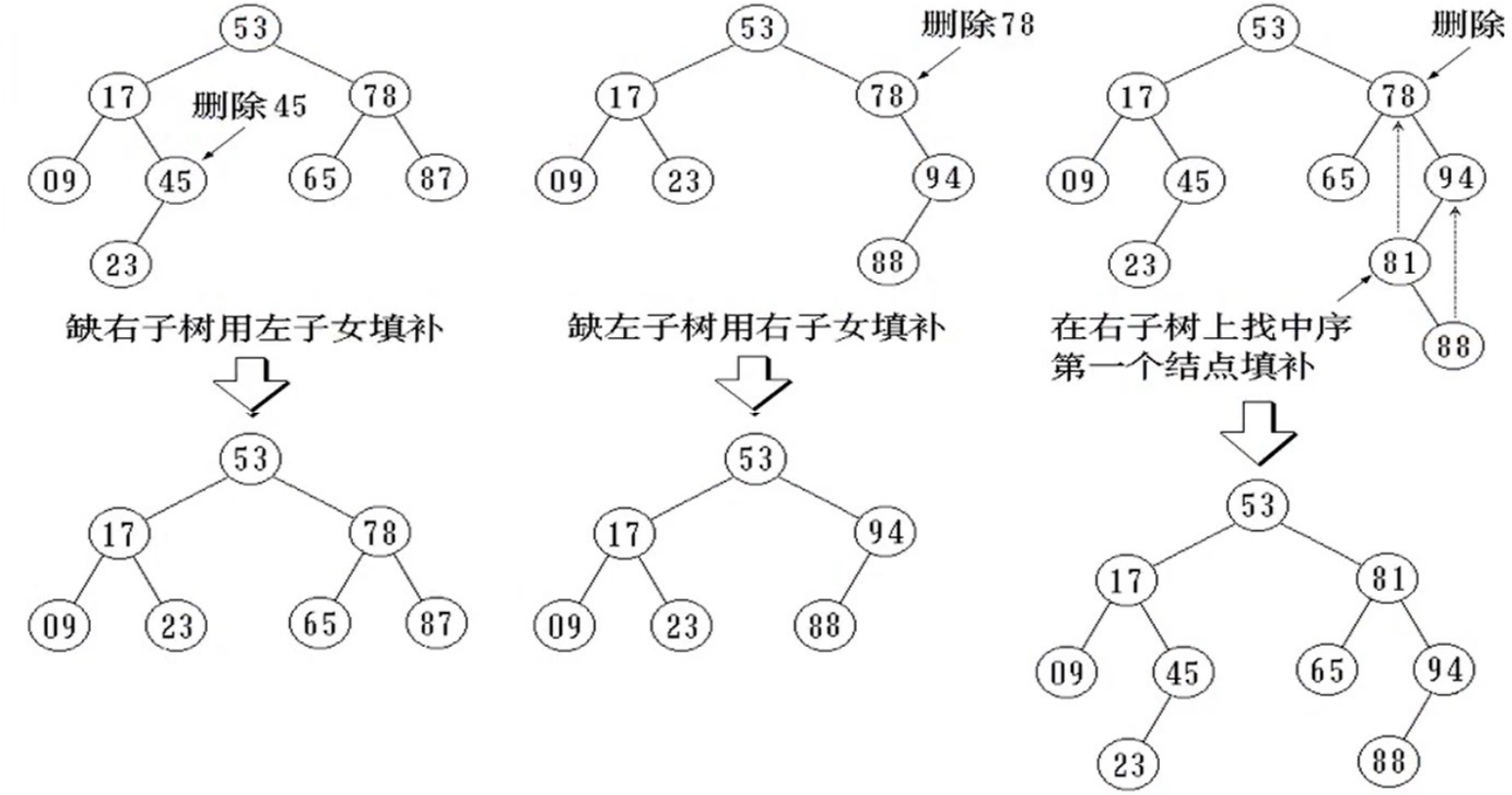
- 以其中序前驱值替换之（值替换），然后再删除该前驱节点。

  前驱是左子树中最大的节点。

- 也可以用其后继替换之（值替换），然后再删除该后继节点。

  后继是右子树中最小的节点。

二叉排序树的操作——删除


53
17
78
09
45
65
87
23
删除45
缺右子树用左子女填补
53
17
78
09
23
65
87
53
删除78
17
78
09
23
94
88
缺左子树用右子女填补
53
17
94
09
23
88
53
删除
17
78
09
45
65
94
23
81
88
在右子树上找中序
第一个结点填补
53
17
81
09
45
65
94
23
88